1994-09

Grain size management in repetitive task graphs for multiprocessor computer scheduling

Negelspach, Greg L.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Grain size Management
in Repetitive Task Graphs
for Multiprocessor Computer Scheduling

by

Greg L. Negelspach

September 1994

Thesis Advisor: Amr Zaky

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1994	Master's Thesis

4. TITLE AND SUBTITLE
GRAIN SIZE MANAGEMENT IN REPETITIVE TASK GRAPHS FOR MULTIPROCESSOR COMPUTER SCHEDULING

5. FUNDING NUMBERS

6. AUTHOR(S)
Negelspach, Greg, L.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(Maximum 200 words)*

Optimal scheduling of parallel programs onto multiprocessor computers is an exponentially hard problem. Because of this, most scheduling algorithms in use today rely on heuristics to determine the best balance of computation and communication costs. However, because of the NP-hard nature of the problem, these heuristics have become very complex.

We are concerned with a specific instance of the problem, throughput scheduling, which aims to optimize the completion rate of repetitive programs, expressed as task graphs, for which the computational and communication needs of the tasks are known in advance. We propose a simpler approach for finding better schedules, which involves testing different grain size modified versions of the task graph to find the one that results in the highest throughput for the given scheduling algorithm. Our heuristic works by alternately fusing or fissioning selected tasks of the graph then evaluating the modified task graph by measuring the expected throughput of its resultant schedule. Because of the generality of this approach, it can be applied to any scheduling algorithm that does not already include grain size modification.

We test the new heuristic using a simulation of the Navy's new standard digital signal processor, the AN/UYS-2, and using various task graphs and scheduling algorithms. We show that this practical approach to scheduling can increase throughput of the Largest Process Time first algorithm by at least 16 percent for our model computer configured with four, eight, or sixteen processors.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Multiprocessor Scheduling, Grain Size Management, Throughput.	56
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

GRAIN SIZE MANAGEMENT IN REPETITIVE TASK GRAPHS FOR MULTIPROCESSOR COMPUTER SCHEDULING

Greg L. Negelspach
Lieutenant, United States Navy
B.S., Oregon State University, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1994

Author: [redacted]
Greg L. Negelspach

Approved by: [redacted]
Amr Zaky, Thesis Advisor

[redacted]
Mantak Shing, Second Reader

[redacted]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Optimal scheduling of parallel programs onto multiprocessor computers is an exponentially hard problem. Because of this, most scheduling algorithms in use today rely on heuristics to determine the best balance of computation and communication costs. However, because of the NP-hard nature of the problem, these heuristics have become very complex.

We are concerned with a specific instance of the problem, throughput scheduling, which aims to optimize the completion rate of repetitive programs, expressed as task graphs, for which the computational and communication needs of the tasks are known in advance. We propose a simpler approach for finding better schedules, which involves testing different grain size modified versions of the task graph to find the one that results in the highest throughput for the given scheduling algorithm. Our heuristic works by alternately fusing or fissioning selected tasks of the graph then evaluating the modified task graph by measuring the expected throughput of its resultant schedule. Because of the generality of this approach, it can be applied to any scheduling algorithm that does not already include grain size modification.

We test the new heuristic using a simulation of the Navy's new standard digital signal processor, the AN/UYS-2, and using various task graphs and scheduling algorithms. We show that this practical approach to scheduling can increase throughput of the Largest Process Time first algorithm by at least 16 percent for our model computer configured with four, eight, or sixteen processors.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The demands on processing performance in many data-intensive fields has generally been greater than the available technology can deliver. In fields such as Digital Signal Processing (DSP), advances in computing hardware technology have quickly found their way to applied problems. Because of this high demand for greater processing power, much effort has been spent on increasing processor performance. As the pace of performance improvements for single processors has slowed, new techniques for providing improvements are required. One such approach is the employment of parallel computing machines. Because they are capable of directly executing the naturally parallel sections of a program, multi-processor computers, or multicomputers offer greater performance over single-processor computers. However, the problem of how to efficiently schedule a program onto a parallel architecture machine has not been definitively solved. In general, algorithms for finding the optimal solutions to most variations of this problem are known to have exponential time complexity, and are therefore intractable for all but trivially small cases. This has lead to the development of mapping heuristics, which attempt to find sub-optimal solutions in reasonable (usually polynomial) computing time.

One tool often employed in the mapping problem is grain-size management, in which tasks of a parallel program may be fused, allowing communication costs to be eliminated between them, or fissioned, to increase the potential parallelism of the program. This is beneficial when the distribution of process execution requirements does not allow for an efficient 'packing' of tasks into a schedule, as may happen if one task is sufficiently 'big' enough to solely determine minimum schedule length, or if there are too many small tasks causing excessive communication. By proper management of the grain size, an optimum trade-off between parallelism and communication overhead can be obtained, facilitating the scheduling process in achieving an efficient mapping.

A. OBJECTIVES

In this thesis, we explore the feasibility of using a new grain-size management heuristic for finding near-optimal schedules on parallel computers. We consider only repetitive programs expressed as task graphs, for which the computation and communication needs of the tasks, and the interaction among them, are static and available *a priori*. Because of their repetitive nature, we are interested in achieving the maximum throughput of program execution, rather than the minimum response time.

We use a new heuristic to manage grain size, which iteratively fuses or fissions selected tasks of the program to find a good grain size for the particular application and multiprocessor architecture. The suitability of a grain-size modified graph is evaluated at each step in the iterative process by actually scheduling it on the target computer, and measuring the expected throughput, defined as program instance completions/unit time. We use two simple scheduling algorithms to keep the time complexity of the new heuristic reasonable. The first is a heuristic which repeatedly places the largest unscheduled task at the end of the processor schedule that affords it the earliest finish time, and the other is a bounded depth-first search algorithm that returns the best schedule found after enumerating a set number of schedules. In each case, throughput is measured by analysis of the scheduled graph.

We test our new heuristic with randomly generated task graphs on a generic multiprocessor computer modeled after the Navy's new standard digital signal processor, the AN/UYS-2 [13]. By using this practical approach to scheduling, we hope to show that high-throughput schedules are achievable without the use of algorithms highly complex in time or space requirements.

B. THESIS ORGANIZATION

Chapter II introduces the necessary background for the scheduling problem, and reviews the relevant work of other researchers in the field. Chapter III discusses the new grain-size management heuristic. Chapter IV contains the methods of, and results for

experiments with the new heuristic, and Chapter IV contains conclusions from the experiments, and suggestions for future work.

II. THE MAPPING PROBLEM

A. DATA FLOW GRAPHS

Mapping a program onto a parallel computer requires first that the sequential dependencies in the program be made explicit. For this, we use Large Grain Data Flow (LGDF) graphs, which are derived from the computation graph model introduced by Karp and Miller [1]. In this model, a program is divided into tasks, or nodes, which are connected by communication arcs. Nodes may represent subroutines or any other part of a programs' computation, such as the parts of a complex algebraic computation. Below is a diagram of a small computation graph:

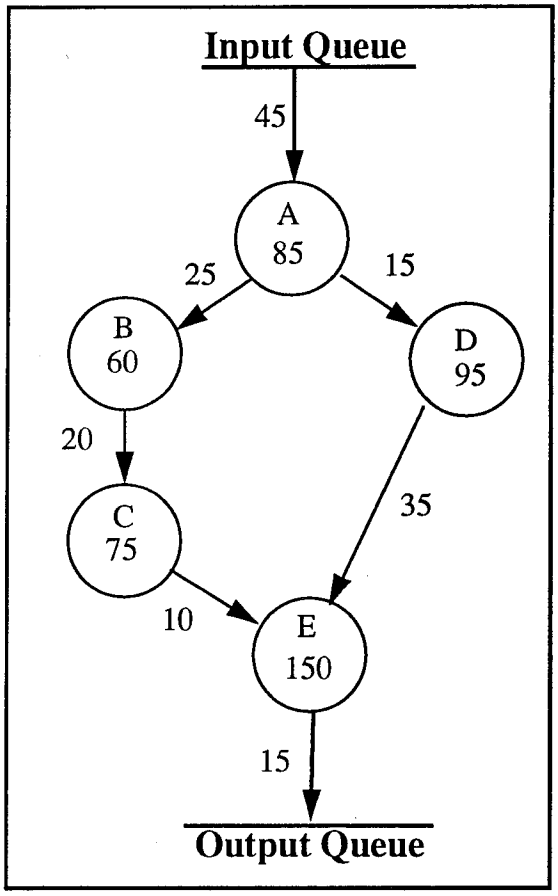

Figure 1: Sample Data Flow Graph

Each program node has been labeled with a letter, and marked with the amount of computation it requires. The arcs represent queues, and the number along each arc represent the amount of data that flows through it for each cycle of the graph. A node is termed a *parent* of another node if it supplies data to the other node, or a *child* of the other node if it receives data from the other node. *Ancestors* and *descendants* are defined analogously if there are intermediate nodes in the connecting path between them.

It is assumed that the computation and communication amounts in the graph are fixed, and known in advance. Communications are assumed to only occur either before or after a node executes, so no inter-process communication or global memory accesses are allowed for executing nodes. Also, there are no undirected arcs; communication direction must always be explicitly specified.

As stated above, we consider only a special case of computation graphs - acyclic graphs - which are more reflective of actual DSP applications, our main emphasis. Additionally, we assume without loss of generality, that graphs have only one 'source' node and one 'sink' node, which solely communicate with the input and output queues. Computation graphs have no such restrictions; however, they may be converted to a functional equivalent in our form by linking all existing source nodes to a new source node with zero computation cost, and linking all sink nodes to a new sink node with zero

computation cost. The figures below show one of these more general acyclic computation graphs before and after conversion to our format:

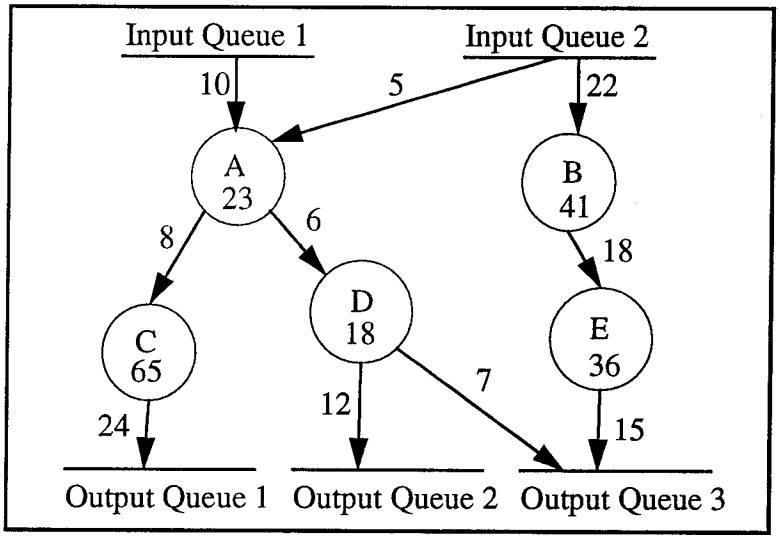

Figure 2: Directed Acyclic Computation Graph

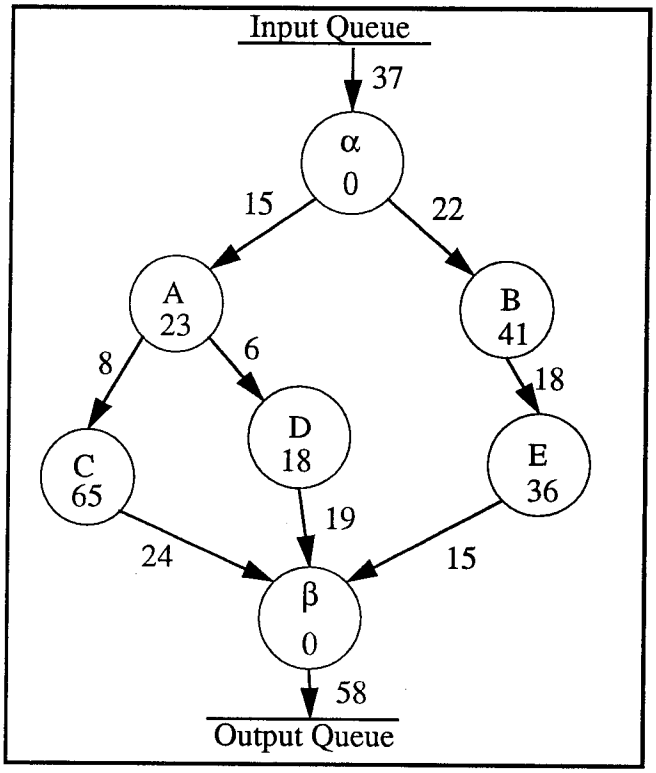

Figure 3: Directed Acyclic Graph in our Format

Our graphs also differ in that we do not model separate *produce* and *consume* amounts on queues. These indicate the actual amount of data written to a queue, and the amount of data removed from that queue, respectively. A difference in these causes the communicating tasks to execute at different rates, or even cause deadlock. For example, a node may write N bytes of data to a queue on every execution, but the node that reads the queue may consume (N + M) bytes per execution. In this case, it takes more than one execution of the writing node to enable execution of the reading node. We therefore only consider task graphs for which the produce and consume amounts for a queue are the same.

Additionally, computation graphs may specify a separate *read* amount for a queue which may be greater than the consume amount. Since a node must have all its data prior to execution, the read amount is also called the queue *threshold*, but usually only if it differs from the consume amount. Thresholds cause extra executions of nodes during the first execution, while queues fill up, but not on subsequent iterations. Unlike consume amounts however, read amounts describe the actual amount of data flow during steady state execution. For this reason, we use read amounts exclusively in our graphs.

B. RELEVENT DATA FLOW GRAPH METRICS

Several aspects of a graph are important to note in the analysis of the mapping problem. The first is graph *width*, which measures the maximum parallelism potential of the graph. It is the maximum number of processors that could work on one instance of the graph at any one time. For the graph in Figure 1, the width is 2; the two concurrently running processes could be (B, D), (B, E), (C, D), or (C, E).

Another is the *critical path*, which is a path in the scheduled graph that incurs the maximum amount of processing time; in general, it need not be unique. The processing time caused by the critical path is known as the *Makespan*. Since it concerns processing time, it includes computation as well as communication costs. Assuming a fully connected architecture, and the convention that one unit of execution takes as long as one unit of

communication, the graph in Figure 1 has a makespan of 485 from the critical path (A, B, C, E).

A graph may also be classified by its *grain size*, which is defined as the ratio of total computation cost to total inter-task communication cost. Graph granularity is often used as a relative term, with the higher ratios designated *large* or *coarse* grained, and the lower ratios, *fine* grained. The term is suggestive of task size, since the granularity of a graph is closely related to the amount of computations done between communications. For the graph in Figure 1, the granularity is 465/165 = 2.82.

C. SCHEDULING

1. Metrics & Methods

The first step in scheduling is deciding which performance metric is the objective of optimization. For graphs not intended to be executed repetitively, minimum *response time* is usually the goal; it measures how long it takes for one instance of the graph to execute. Here, the critical path determines the minimum length of a schedule. Figure 4 below is a *Gantt* chart which shows the minimum-response time scheduling of the graph in Figure 3 for a four processor machine.

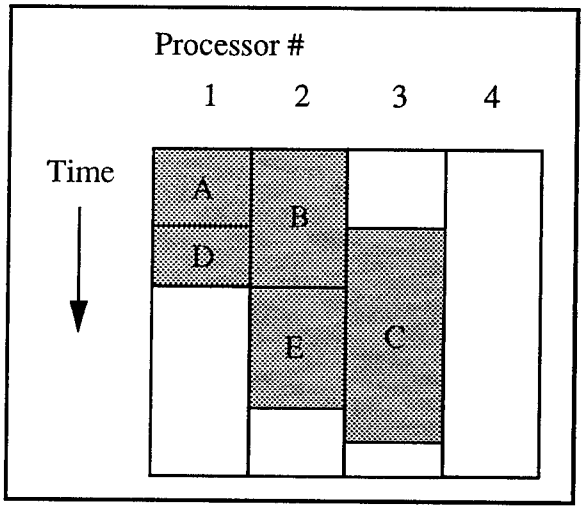

Figure 4: Gantt Chart of Graph in Figure 3

For this example, communication costs are assumed to be zero. Note that schedule length is affected by the precedence constraints between tasks, as well as the number and size of the tasks, and number of processors available. The response time for this schedule is 88, due to the critical path (A, C). When communication costs are taken into consideration, it is usually assumed that such costs are eliminated between tasks executing sequentially on the same processor. If this is the case, then Highest Level First (HLF) algorithms may be used, which reduce response time by successively scheduling the two most intercommunicating nodes in the critical path to the same processor. Reference [4] surveys some of the more prevalent algorithms.

For graphs intended to be executed repetitively, maximum *throughput* is the target of optimization, which measures the rate at which graph instances complete. Makespan reduction is not pursued here because there is a better way to increase throughput - by graph pipelining. The idea is to pack the nodes into the schedule where they fit best, without regard to precedence constraints. Causality is maintained by assigning out of order tasks to previous instances of the graph. For heavily pipelined graphs, several instances of the graph may be executing at any one time. The next figure shows a maximum throughput schedule for the same graph as in the above example. Communications are again ignored for simplicity.

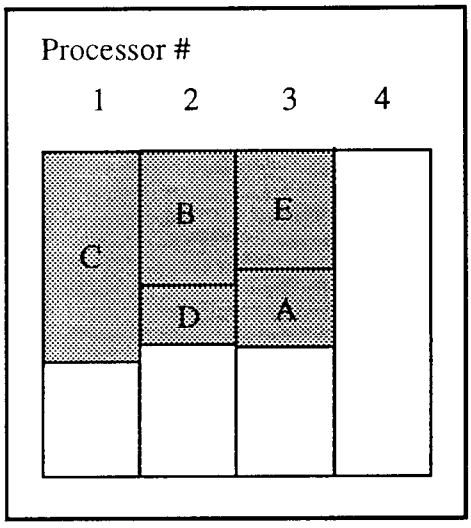

Figure 5: Scheduling of Graph in Figure 3 for Maximum Throughput

It is important to note the difference between the schedule, which depicts processor assignments in time, and the resultant execution pattern of each graph instance. Scheduling nodes out of order causes execution patterns that look different from that which is shown in the Gantt chart.

For example, in the schedule of Figure 5, nodes 'C', 'D', and 'E' can not possibly execute on the same instance of the graph as nodes 'B' and 'A', since they are shown starting before the completion of the nodes that supply them input data. To make the schedule work correctly, nodes 'B' and 'A' need to work on a set of data ahead of the other nodes. Figure 6 shows how the above schedule would actually start execution on our four-processor computer example:

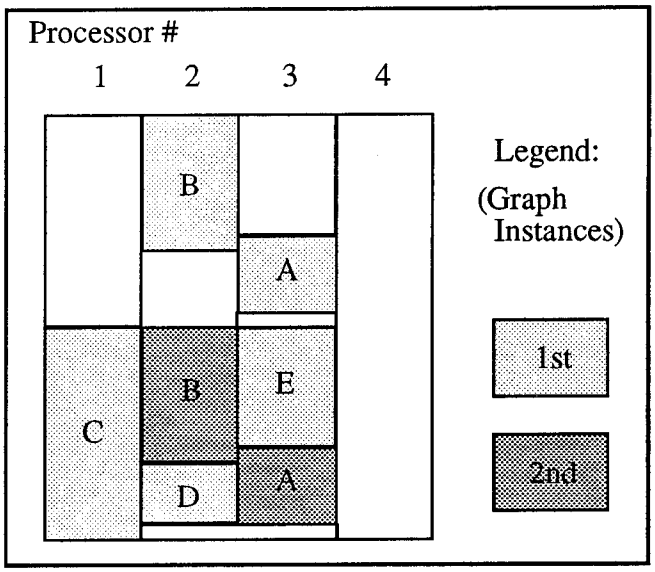

Figure 6: Execution Sequence of Max. Thoughput Schedule

Tasks 'C', 'D', and 'E' are not shown as part of the schedule in the first iteration because no data is available for them yet. On the next and subsequent iterations, all nodes have data to execute, but 'C', 'D', and 'E' remain one instance behind nodes 'A' and 'B'. Thus processor assignments look like the Gantt chart at each schedule iteration, but individual graph instances each appear as shown in Figure 6.

The reason pipelining is better for increasing throughput, is that it eliminates the processing delays due to precedence constraints. This makes it much easier to find shorter schedules. In the previous schedule, graphs would complete at the rate of one instance every 88 cycles, giving a throughput of 1/88 instances/cycle, while the new schedule would give a throughput of 1/65 instances/cycle. The increase in throughput however, has come at the cost of increased response time. It must now be computed by summing the time required to complete the first iteration of the schedule, which equals the makespan of nodes 'B' and 'A' (which is scheduled to be the same as the makespan for 'C'), and the time required to complete nodes 'C', 'D' and 'E' in the second iteration of the schedule (which again equals the length of 'C'). Thus, it is now 130, up from 88 of the previous schedule.

It is also important to note that the throughput of a schedule does not depend on the length of the schedule as a whole, but on the maximum length of its constituent processor schedules instead. Consider Figure 7 below, in which the previous schedule has been modified by moving tasks 'B' and 'D' down relative to the other tasks:

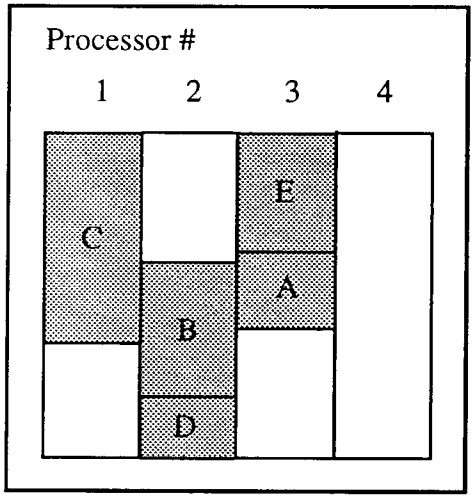

Figure 7: Modified Schedule

The schedule looks longer, but the throughput remains the same, since the individual processor schedules have not changed:

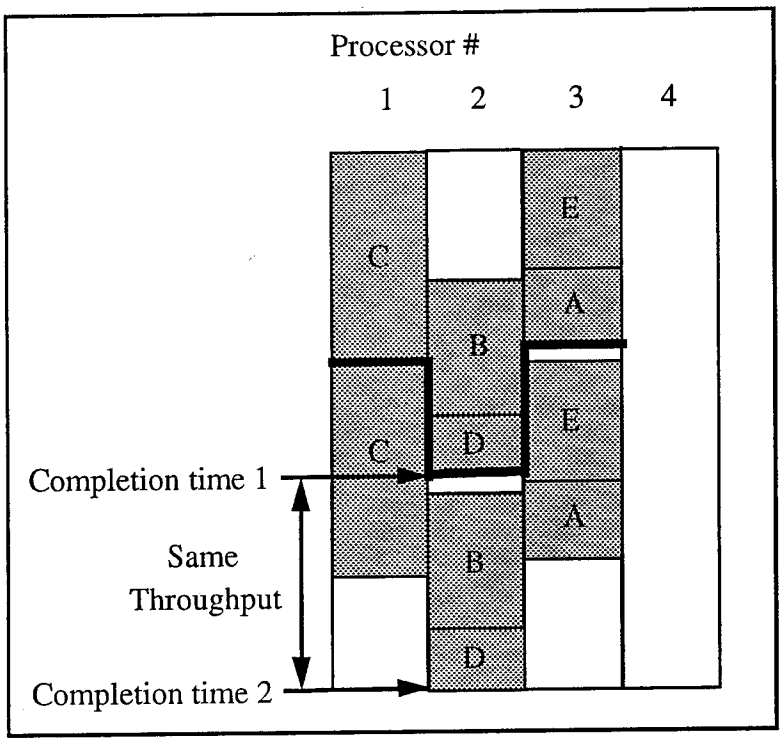

Figure 8: Throughput Analysis of Modified Schedule

2. Types of Scheduling Algorithms

Scheduling algorithms may use any number of different techniques to balance schedule performance and computational complexity. We make a brief survey of the more prevalent types here, and note their applicability to our particular scheduling needs.

a. Optimal Algorithms

These algorithms perform a state-space search of the possible solutions to the scheduling problem in order to find the best one. Unfortunately, it is not possible in the general case to know ahead of time if a given schedule is the optimum. Thus it is necessary to consider all possible schedules. Numerous researchers have shown this to be NP-hard for most variations of the problem. Because of this, optimal scheduling algorithms are not often used in practice. One notable exception in this area is the work using the A* search approach by Shen and Tsai in [9].

b. Heuristic Algorithms

Heuristic algorithms avoid exponential time complexity by introducing simplifying assumptions about the solution space which enable the search to proceed in more promising directions. Generally, they start with an initial schedule, then iteratively try to improve it, using the assumptions to help guide the search. An algorithm may search progressively, incrementally improving the solution until a local optimum is found, or probabilistically, searching the solution space regions most likely to contain the optimum. They are very widely used, because of their unmatched combination of low time-complexity and high schedule quality.

c. Partitioning Algorithms

Partitioning algorithms are heuristics that assume optimum schedules may be obtained by finding the assignment of tasks to processors that minimizes inter-processor communication costs. This approach, however, does not make use of information on task execution requirements or precedence constraints. Thus, it is more applicable when such information is not available, as in a distributed computing environment. Work in this area includes techniques by Kirkpatrick, Gelatt, and Vecci in [10]; the work by Pothen, Simon, and Liou in [7], known as the *spectral method*, or *recursive spectral method* (RSB), and the much-referenced work by Kernighan and Lin in [8]. Thomae gives an overview of the different partitioning techniques in [6].

d. Throughput-Specific Algorithms

The simplest algorithm for high throughput schedules is to assign each processor a complete copy of the task graph, and synchronize them in a staggered fashion to ensure even throughput. This technique does not cause any interprocessor communication costs, and every processor can be fully utilized, resulting in linear speedup. The approach, however has several potential drawbacks. First, it assumes that local processor memory capacity is large enough to store an entire task graph. Second, not every processor may have access to an input/output processor. Thirdly, the scheme does not

support tasks needing access to previous instances of data, and modifications to include such capability would incur substantial communication costs. For these reasons, and the fact that my thesis advisor told me it was too simple to get a good thesis out of, we do not consider this approach.

A more common method of generating high throughput schedules is through the use of *List Scheduling* algorithms. These are discussed in the literature in the context of scheduling independent, non-communicating tasks, but can also be applied to throughput scheduling if communication costs can be fully included in the computation cost of a task, a possibility if communication costs can be assumed to be constant, and independent of the assignment of tasks to processors.

List scheduling algorithms work by first ordering all the tasks in a list, then successively assigning them to the processor with the earliest finish time. Garey, Graham, and Johnson have shown in [2], that if the ordering is random, such an algorithm will produce a schedule with a graph instance completion rate of less than or equal to (2 - 1/M) of the optimum completion rate, where 'M' is the number of processors. For large M, this bound approaches 2. They state that such algorithms exist with time complexity of Nlog(M), where 'N' is the number of tasks to be scheduled.

They discuss a second algorithm based on list scheduling, called LPT (largest process time), in which the tasks in the list are sorted in decreasing execution size order first. They show that for this algorithm, the bound can be reduced to (4/3 - 1/(3M)) * optimum, which for large M, approaches 1 1/3. This can be accomplished with an increase in time complexity to only Nlog(MN).

A third algorithm is described, called MULTIFIT, that performs a binary search on the minimum possible completion time, using the LPT algorithm above. This approach, it is claimed, produces schedules within a factor of 1.22 of the optimum for all M, with a time complexity proportional to Nlog(MN).

As stated before however, all these algorithms assume no, or constant communication costs between tasks. Because of this, they are applicable only for problems

involving computation intensive programs, and computers with significant communication capabilities, i.e., those with fully connected architectures, or the capability to overlap communication with computation.

Hoang and Rabaey in [3] describe a scheduling approach similar to the MULTIFIT algorithm, except they successively schedule the task with the greatest difference between its earliest and second earliest start time among feasible processor assignments. Communication costs are modeled by using an architecture specific function which gives the earliest start time for a task given the current schedule. They also assume hierarchical task graphs, so that *bottleneck* nodes, those that solely determine schedule length, may be decomposed into their elemental parts.

They give no upper bound on the throughput performance of their algorithm, but they show impressive speedup improvement over other algorithms, and show that it runs in at least O(N(N + E)) time, where 'E' is the number of edges in the graph.

3. Additional Factors Affecting Schedule Length

Until now, the only aspect of computer architecture we have considered in depth in the scheduling process is the number of available processors. However, a computer's communication capabilities can be at least as important in determining maximum performance. We now focus on these remaining communication aspects of computer architecture that affect the scheduling process. There are three to consider:

a. Communication Distance

This is the sum of the delays needed to send data between processors and global memory modules via the communication, or interconnection network. It is known as a distance, because it is dependant only on the path taken. However, paths may be determined at run-time, and need not remain the same for any processor and memory module combination.

b. Memory Contention

Memory contention refers to the delay in a memory reference due to a logical or hardware restriction. Logical restrictions enforce that only one write operation may occur to a memory location at any one time. Hardware restrictions involve limits on the number of simultaneous read accesses allowed to a memory location or the block that it resides in. Hardware restrictions may range from none at all, to as severe as restricting access to the entire memory module to one request at a time.

c. Link Contention

This covers all the delays due to bottlenecks in the processor-to-memory interconnection network. Fully connected architectures have no such restrictions, but such networks are impractical for systems with large numbers of processors and memories, since the number of connections required grows as the square of the number of destinations. Instead, they rely on neighboring processors to route data between processors not directly connected, or employ an interconnection network, which is dynamically managed to prevent collisions of data or requests for data. Link contention occurs when the network must delay messages because to prevent such collisions, or in the case of the routing processors, when a processors' communication limits have been reached.

D. GRAIN SIZE MANAGEMENT

1. Fusion

Fusion is a task graph modification technique to aid the scheduling process in which selected nodes are combined to reduce communication costs. We use this term to denote the actual merging of tasks into one, and distinguish it from the term *clustering*, which refers to the assigning of distinct tasks to a processor. Our fused nodes subsume all the computation and communications of their included nodes, and replaces them in the graph. The computation cost of a new node is defined to be the sum of its member nodes' costs, and its communication arcs to be the union of its members arcs, with redundant arcs

combined if necessary. The figure below shows an example of fusing for a section of a task graph:

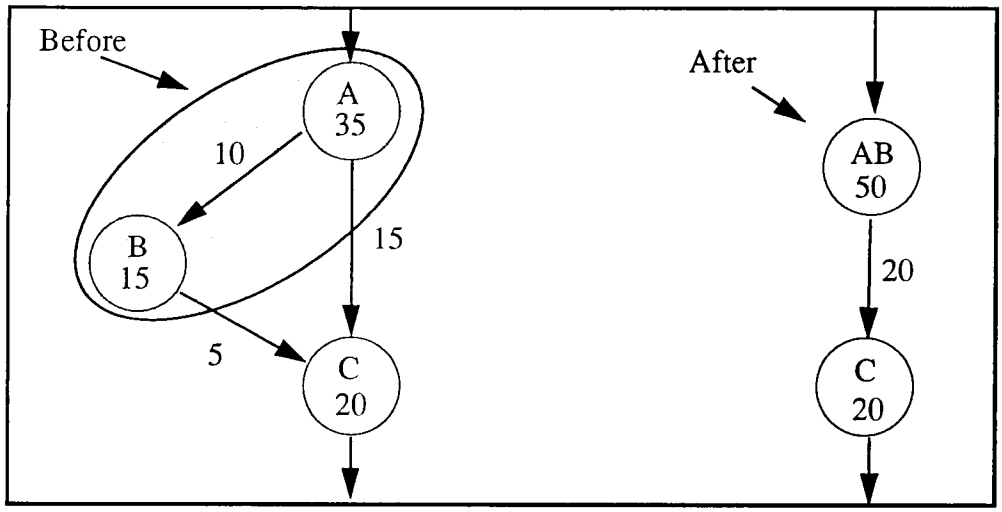

Figure 9: An Example of Node Fusion

When tasks in a graph are fused, inter-task communications decrease, but total computations do not change, thus reducing granularity and communication overhead for the graph. Since we deal only with acyclic graphs however, we must be careful to avoid the creation of cycles. Notice what happens when nodes 'A and 'C' are fused in the following graph instead:

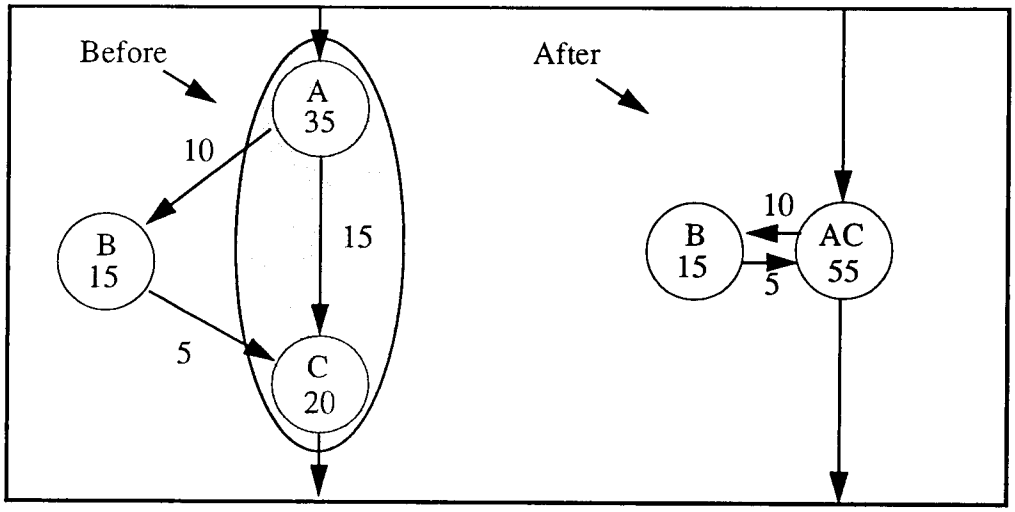

Figure 10: Fusion-Induced Cycle

Node 'B', which has both parents and children in the fused node, causes a cycle. To avoid this problem, we include all those nodes in the graph that are both an ancestor and descendant to the two nodes being fused. In general, if α and β are the two nodes to be fused, and we know that α is an ancestor to β, then we need to include all the intermediate nodes which may pass data from α to β. Figure 11 below illustrates the situation:

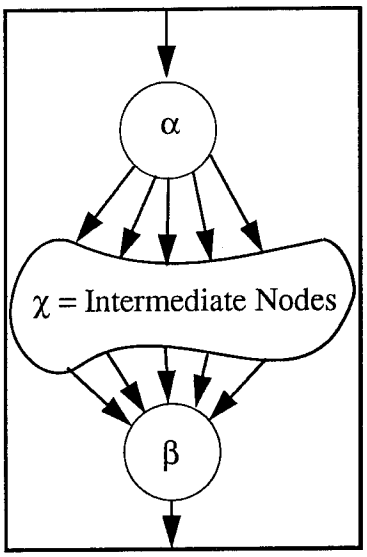

Figure 11: Inclusion of Intermediate Nodes in Fusion

If χ is the set of nodes to include in the new cluster, then it can be given by the formula $\chi = \{\alpha\} \cup \{\beta\} \cup (Decendants(\alpha) \cap Ancestors(\beta))$. If α and β are unrelated (neither is an ancestor or descendant of the other), then the formula still holds, but the intersection is empty, and the set consists of just α and β.

The main advantage of fusion is that it can be used to reduce communications, and thus improve performance. However, it also simplifies the scheduling process. For example, consider the scheduling of a task graph that contains a small region of communicating nodes with low granularity. If fusing two nodes does not produce a node that is too large, then it is almost certain that any good scheduling would put them both on the same processor to eliminate communication costs. Thus, it is unnecessary to consider

them separately. The advantage for scheduling results from the reduction in task graph size, which enables faster schedule production, or the use of better, higher-time complexity algorithms.

The use of fusion must be carefully managed however, as injudicious use can cause inefficient schedules. Overuse may restrict the choices available to the scheduling algorithm enough to prevent an efficient packing of tasks into a schedule, or may even result in nodes whose processing times are greater than that of the original scheduled graph. Thus, there is an optimum level of granularity that exists between each task graph - computer combination.

2. Fission

In this thesis, we also consider *fissioning* of nodes, which is essentially the opposite of fusing. Fissioning a node that has been fused returns that region of the graph back to its previous, un-fused state. Thus, graph nodes may be hierarchically defined.

Node fission is used primarily when a single node is large enough to solely determine schedule length. In such cases, fissioning enables the offending node to be broken down into smaller, more efficiently scheduled pieces. As with fusion however, the process must be carefully considered.

For nodes that have not been previously fused, we make the assumption that fissioning creates two new nodes, each with half of the original nodes' amount of computation, and connected with a single queue with a produce-consume amount equal to the sum of the original nodes' input and output queues. We make this conservative assumption in the absence of prior knowledge about node fission, but the matter warrants further research.

E. MAPPING FOR DATA FLOW COMPUTERS

In this thesis, we evaluate our heuristic using a computer model based on the data flow computing paradigm. We describe this model now, and discuss the aspects of it relevant to the mapping problem.

1. Data Flow Computers

Data flow computers are a type of MIMD architecture machines specially designed to implement computation graphs directly. They employ a hardware scheduler to control task execution at run-time which keeps track of queue status for all nodes in the graph being executed. When all a nodes' input queues have reached their threshold, it is put in a ready queue in the scheduler, along with all the other nodes waiting to execute. As processors becomes free, the scheduler causes nodes in the ready queue to be loaded onto waiting processors, along with the data, called *tokens*, from their input queues. When a node completes execution, the scheduler is notified, and their output tokens are written to the output queues. The scheduler is then updated on the status of the queues, and the cycle continues until there are no more nodes in the ready queue.

The chief difference from more conventional, controlled flow multiprocessor architectures is that program execution is implicitly controlled by the computer itself. On other MIMD machines, the programmer (or compiler) must specify which tasks will execute on which processors, and to which memory modules they will communicate with. However, the advantage of programming simplicity for data flow computers may come at the expense of efficiency; there is much potential for overhead associated with the run time control of multi-task program execution.

To alleviate some of the overhead, data flow machines can be made to have the capability to overlap communication and computation. This is achieved by providing every processor with its own communication co-processor to handle data requests between the processor-local memory and global memory. The scheduler is then able to load ready nodes and their data to processors when their communication processor is idle, even though the computation processor might be busy processing a node.

2. RC Scheduling

A technique for further reducing the overhead of data flow computing was introduced by Shukla, Little, and Zaky in reference [11]. They introduced the RC

scheduling algorithm, which can control the runtime behavior of repetitive task graphs on data flow computers. This is significant, because task execution order in a data flow computer is dependant only on the availability of data, and thus may not always lead to the highest throughput. RC scheduling provides a way to enforce a schedule, so that throughput remains consistent and predictable. The technique involves selectively adding queues and/ or queue parameters to a task graph to ensure that tasks become ready for execution at predictable times, forcing execution to follow the desired order. In reference [12], Cross demonstrates its effectiveness for various test graphs and computer configurations.

F. THE EMSP COMPUTER

Experiments for our new heuristic are evaluated using a model of the Navy's new standard digital computer, designated the AN/UYS-2, or EMSP for enhanced modularity signal processor. It is a Large Grain Data Flow (LGDF) processor, which means that it conforms to the data flow paradigm only for large tasks. At the instruction level, it is a conventional, controlled flow multiprocessor computer, capable of executing multiple processes per processor concurrently. This enables the use of more efficient, conventional programming techniques for processing within a task, while allowing data flow control to maximize the parallelism at the inter-task level.

It consists of five modular sections, known as functional elements (FE). These are the arithmetic processors, global memories, the scheduler, I/O processors, and the data bus. The bus can be configured with either 8 or 16 ports which provide connections for the other modules to connect with the bus. Up to four modules may be connected to a port using a *concentrator*, which acts as a sub-bus between the main bus and the collection of modules. Reference [13] gives a comprehensive review of its architecture.

III. THE GRANULARITY MANAGEMENT HEURISTIC

A. MOTIVATION

Because of the high complexity of the problem, most scheduling algorithms in use today rely on heuristics to determine the best balance of computation and communication costs. These heuristics have become very sophisticated in order to provide the highest schedule quality for the given run time complexity. In light of the nature of the problem, it is unlikely that simpler methods exist for finding better schedules in time comparable to the algorithms available today. Additionally, there is little available insight on how to improve on current methods. Therefore, instead of inventing a more sophisticated heuristic, we have chosen a more pragmatic approach for improving the scheduling process - grain size management.

B. THE HEURISTIC

Our approach for improving the scheduling process is to find the best combination of grain size for the given task graph and processor architecture. We do this by using an iterative process that heuristically chooses nodes in the graph to fuse or fission, then evaluates the modified task graph by measuring the expected throughput. Expected throughput is measured by using the supplied scheduling algorithm, or even by simulation of the scheduled graph. Since a specific scheduling algorithm is not specified, this method will work for any scheduling algorithm that does not already modify task granularity.

The heuristic is called GSM, for Grain Size Management. It first finds all the fusions that improve expected throughput, then all the fissions. This cycle is then repeated until no improving grain size modification can be found. Pseudo code of the algorithm is shown below:

```
function GSM_Schedule (G : in Process_Graph) return Schedule is

    Worked : Boolean;

begin
    Repeat
        Try_Fusing (G, Worked);
        Try_Fissioning (G, Worked);
    Until not Worked;

    return Some_Schedule (G);
end GSM_Schedule;
```

Figure 12: GSM Scheduling Heuristic

Function Some_Schedule is global to this function, and is used by both sub-procedures.

The procedure Try_Fusing works by successively fusing the two nodes of the graph which share the largest communicating arc. As described before, fusing includes all the necessary nodes to prevent cycles. It then schedules the graph, using the global scheduling algorithm, and determines the expected throughput. If the throughput increases, Try_Fusing starts over again, with the newly fused graph. If throughput does not increase, then the fusing is undone, and the next two nodes are considered. If no fusions that improve the schedule can be found, then the procedure returns with a fail flag (not 'Worked'). The figure below shows the pseudo code for the procedure:

```
procedure Try_Fusing (G      : in out Process_Graph;
                      Worked : in out Boolean) is


   procedure Find_Fusion (G         : in out Process_Graph
                          Found_One : in out Boolean) is

      Q : Priority_Queue := Edges (G);
      E : Edge;
      L : Natural        := Length (Some_Schedule (G));

   begin
      Found_One := False;
      while not Empty (Q) and not Found_One loop
         E := Pop (Q);
         Fuse (E, G);
         if (Length (Some_Schedule (G)) < L) then
            Found_One := True;
         else
            Undo_Last (G); -- Restores Graph to its prior state.
         end if;
      end loop;
   end Find_Fusion;

begin
   Repeat
      Find_Fusion (G, Worked);
   until not Worked;
end Try_Fusing;
```

Figure 13: Try_Fusing Procedure

Procedure Try_Fissioning is similar to Try_Fusing, except that it tries fissioning the nodes in decreasing size order. It is shown below:

```
procedure Try_Fissioning (G       : in out Process_Graph;
                          Worked  : in out Boolean) is


   procedure Find_Fission (G         : in out Process_Graph
                           Found_One : in out Boolean) is

      Q : Priority_Queue := Nodes (G);
      N : Node;
      L : Natural        := Length (Some_Schedule (G));

   begin
      Found_One := False;
      while not Empty (Q) and not Found_One loop
         N := Pop (Q);
         Fission (N, G);
         if (Length (Some_Schedule (G)) < L) then
            Found_One := True;
         else
            Undo_Last (G); -- Restores Graph to its prior state.
         end if;
      end loop;
   end Find_Fission;

begin
   Repeat
      Find_Fission (G, Worked);
   until not Worked;
end Try_Fissioning;
```

Figure 14: Try_Fissioning Procedure

Essentially, our method is simply a greedy algorithm that uses the expected throughput of the scheduled graph as the objective function. We make no claim on its run-time complexity, since the number of nodes in the input graph can grow arbitrarily during the Try_Fissioning procedure, especially if the amount of computation has little effect on schedule length.

IV. EXPERIMENTS

A. METHOD

Testing of the GSM algorithm was conducted by simulation using a randomly generated set of 60 node, 120 edge task graphs. Graph node and edge weights were generated randomly using a uniform distribution in (0.0, 10.0], and (0.0, 2.0] respectively. The connectivity of graphs was also generated randomly, although this method required that additional edges be added to ensure that graphs would be connected, and have exactly one source node and one sink node. Thus, 120 is a lower bound on the number of edges.

The first series of tests were conducted to compare the throughput performance of the Largest Process Time first algorithm, LPT and GSM-LPT algorithm on the AN/UYS-2. A set of 100 graphs was used, with an initial grain size normally distributed about 2.5. A schedule was generated for each graph and each combination of algorithm and machine configuration (4, 8, or 16 processors). Throughput was determined by analyzing the generated schedule, which included costs for computation and communication delays due to the amount of data; however, delays due to communication distance or link or memory contention were not considered.

B. RESULTS

1. GSM-LPT vs. LPT

Figure 15 shows the speedup verses initial grain size of each graph scheduled by the LPT algorithm. The plot shows that the algorithm has very predicable, but not impressive performance:

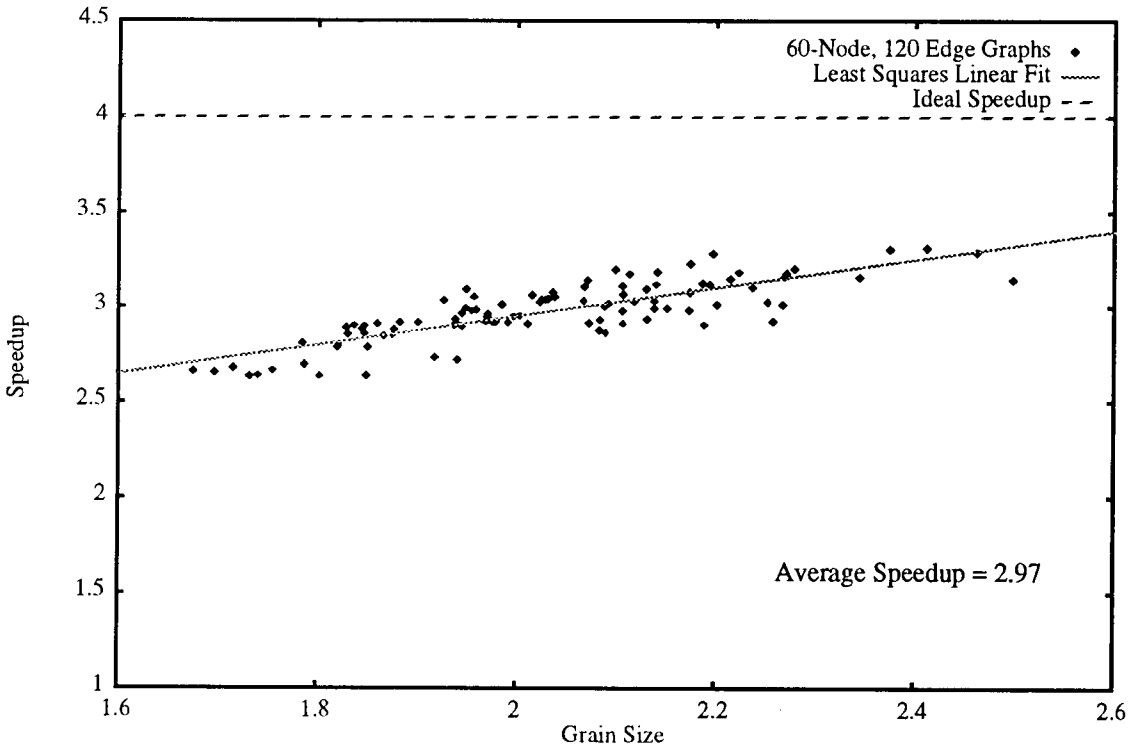

Figure 15: Speedup of LPT Algorithm on 4-Processor AN/UYS-2

As would be expected, speedup increases with higher granularity, because of reduced communication overhead. Using the least squares linear fit, the relationship is Speedup = 0.756 * Initial Granularity + 1.435. The average of all data points gives a speedup of 2.97.

The next graph shows the performance of the GSM-LPT algorithm for the same four processor architecture:

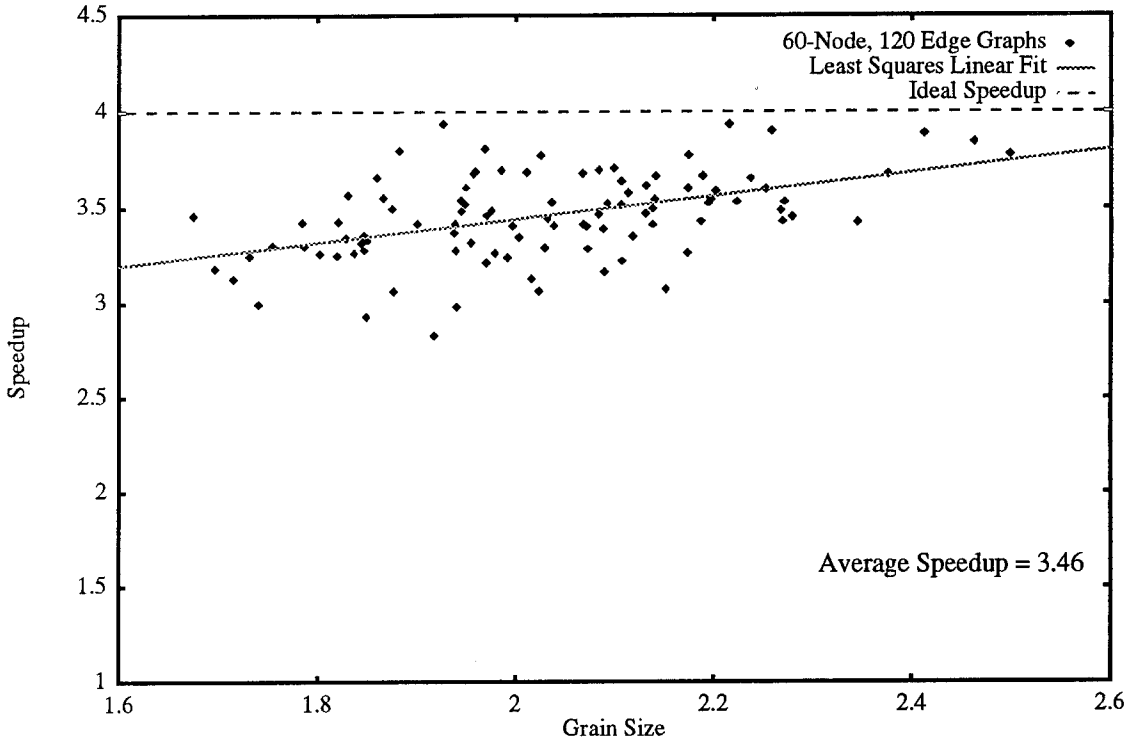

Figure 16: Speedup of GSM-LPT Algorithm on 4-Processor AN/UYS-2

Speedup is higher than for the straight LPT algorithm, although the performance varies more from the average. In each case the GSM-LPT algorithm outperforms the LPT algorithm. For this algorithm, the linear approximation to speedup performance is given by Speedup = 0.608 * Initial Granularity + 2.224. Averaging over the all the available data points gives an average speedup of approximately 3.46, about 16.5% better than the straight LPT algorithm.

The next plot, Figure 17, shows the performance of LPT on an 8 processor AN/ UYS-2. Speedup is proportional to the 4 processor results, although there is more variance in the data:

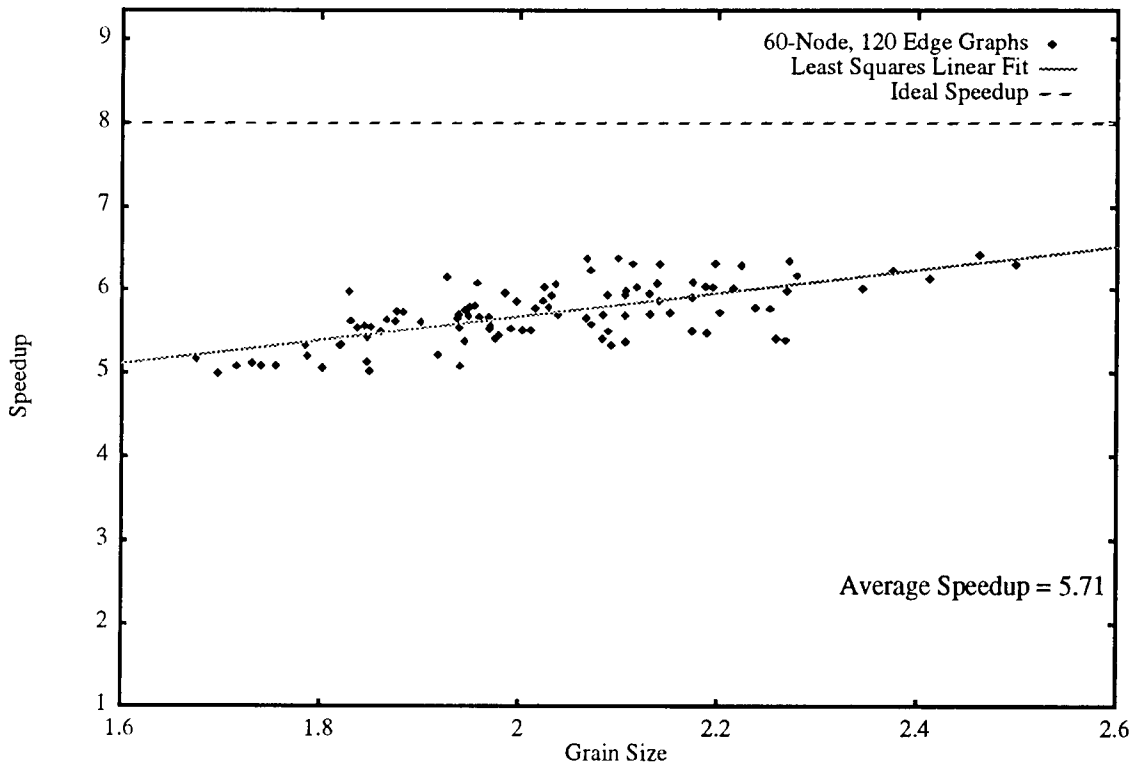

Figure 17: Speedup of LPT Algorithm on 8-Processor AN/UYS-2

The linear approximation to speedup in this case can be given by the formula Speedup = 1.418 * Initial Granularity + 1.832. While this seems much better than for the four processor case, it must be noted that the formula would have to be a greater by a factor of two over the four processor case just to have the same relative throughput. For this eight processor data, the actual factor is just less than two. The average of all speedup values in this plot is 5.71.

The performance of the GSM-LPT heuristic on 8 processors is shown next in Figure 18:

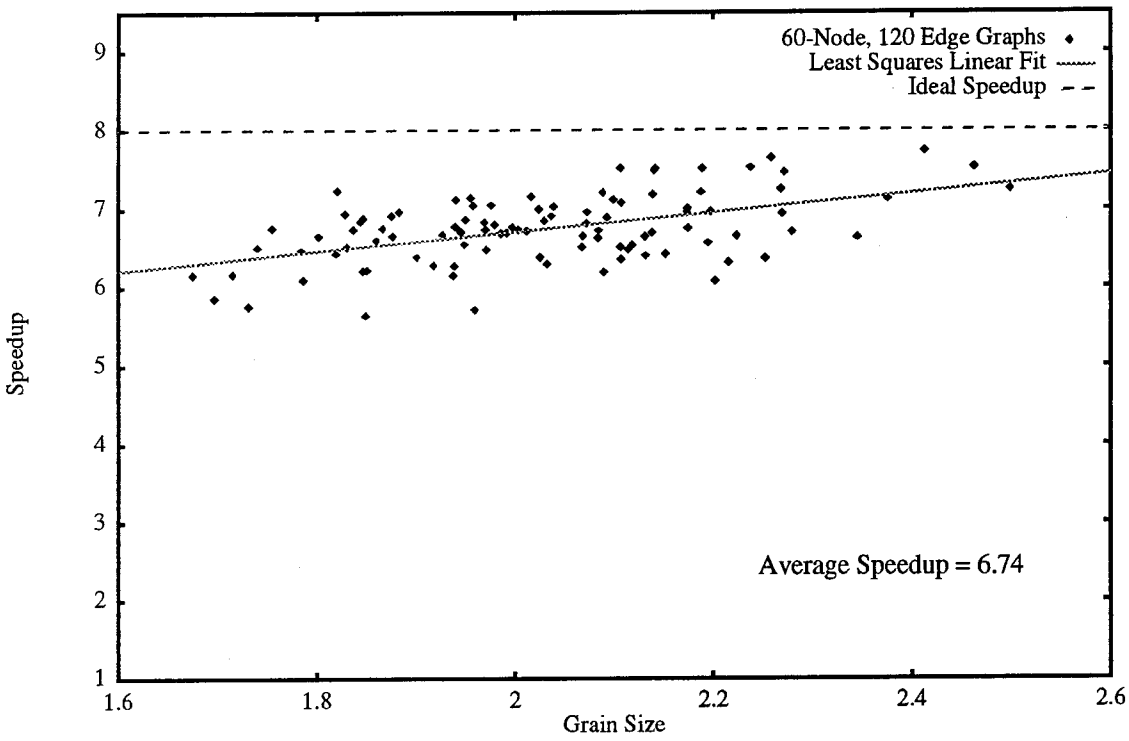

Figure 18: Speedup of GSM-LPT Algorithm on 8-Processor AN/UYS-2

Here, the performance improvement over the straight LPT algorithm is similar to that seen in the four processor comparison. Speedup is approximated by the formula Speedup = 1.243 * Initial Granularity + 4.217, and average speedup is 6.74, 18.0% better than for straight LPT. Again, as in the four processor case, speedup is greater for GSM-LPT than LPT alone, but there is more variance in the speedup for GSM-LPT.

The next two plots show the results of the same comparison for the 16 processor case:

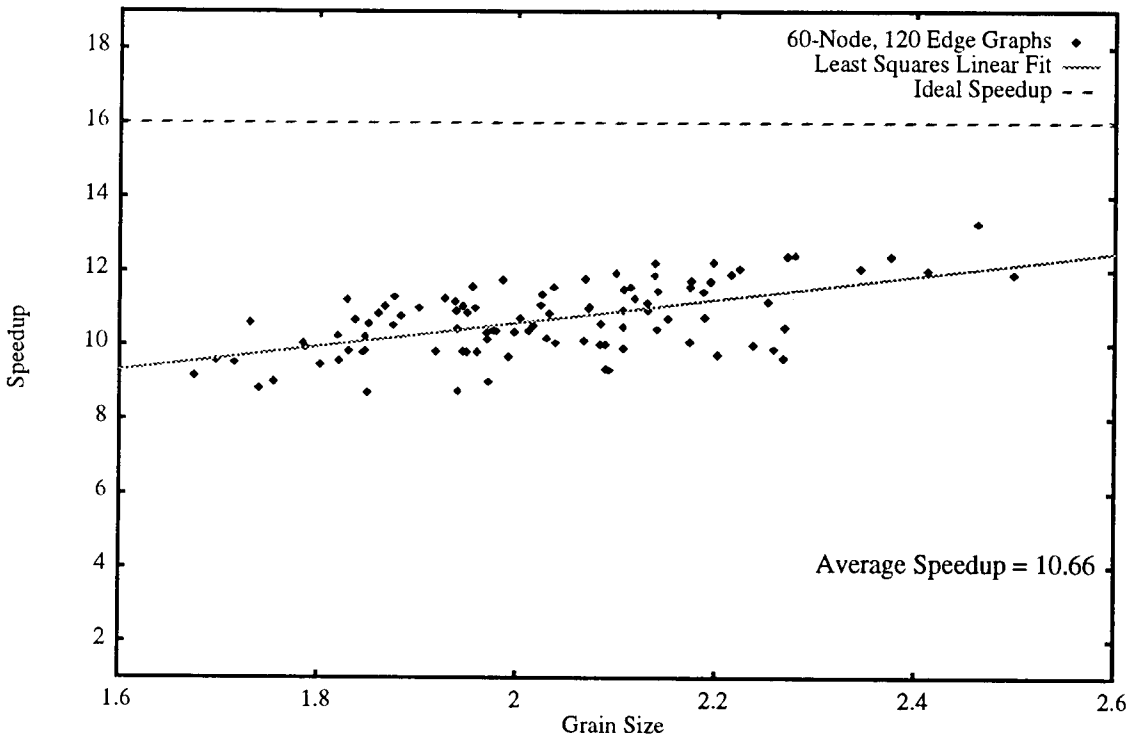

Figure 19: Speedup of LPT Algorithm on 16-Processor AN/UYS-2

The performance of the LPT algorithm in the 16 processor case is very similar to the eight and four processor cases, except speedup varies more. The relationship between speedup and granularity is approximated by Speedup = 3.165 * Initial Granularity + 4.240. Average speedup over all values in the plot is 10.66.

The last plot for this series of experiments shows the performance of the GSM-LPT heuristic for the 16 processor case:

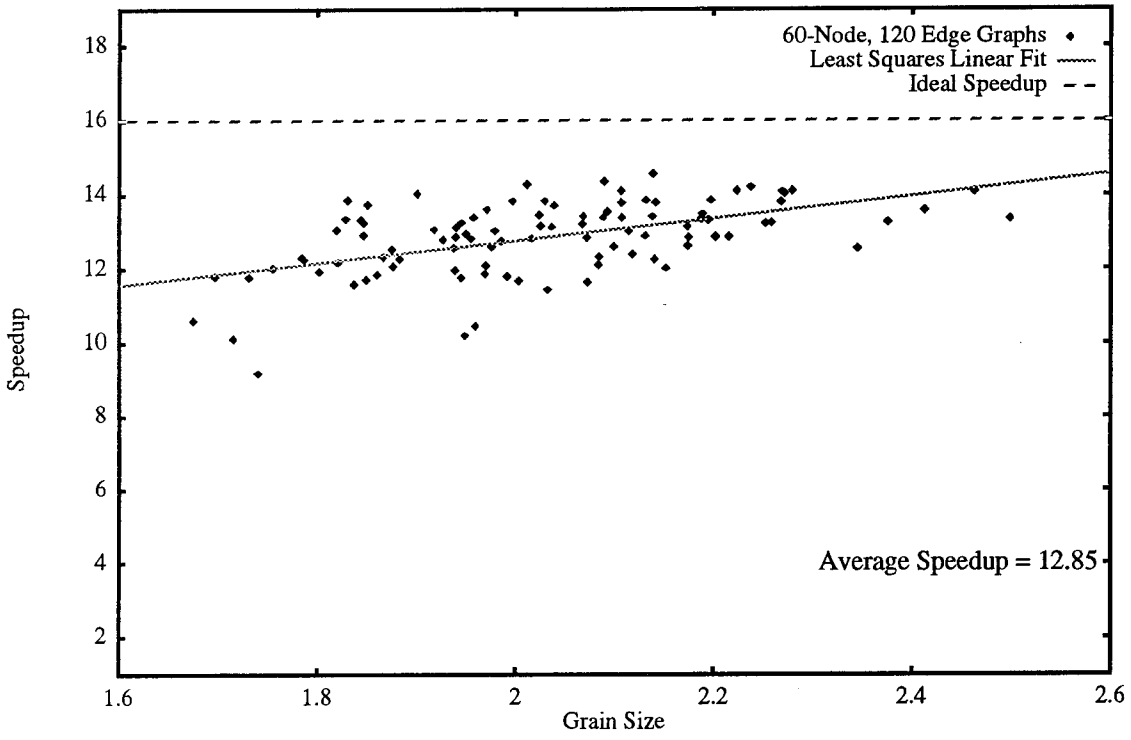

Figure 20: Speedup of GSM-LPT Algorithm on 16-Processor AN/UYS-2

In this plot, we see the largest variance for speedup among all the tests conducted, however for each data point, the speedup for the GSM-LPT algorithm is greater than that for the LPT algorithm. The linear approximation to speedup performance is given by Speedup = 2.999 * Initial Granularity + 6.764. Overall speedup is 12.85, a 20.5% improvement over straight LPT.

For each of the preceding six figures, it is interesting to note that in every case, the average improvement in speedup is approximately equal to average of the LPT performance and ideal performance. Thus while improvements in actual speedup are modest, the percent increase in possible increase are significantly more substantial.

The next result, Figure 21, summarizes the data from the six previous figures by processor configuration and scheduling algorithm:

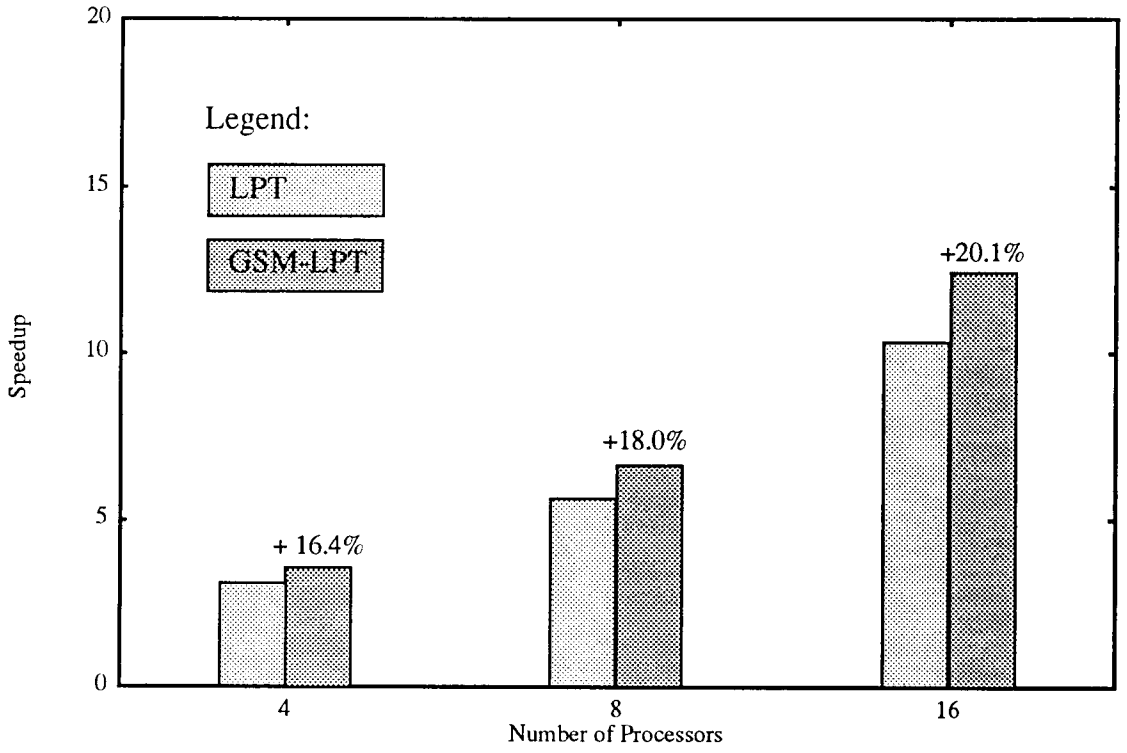

Figure 21: Throughput Performance of LPT and GSM-LPT Algorithms

Note that the increase in speedup performance actually improves with the number of processors to schedule. This is possibly due to the advantage GSM-LPT has over LPT in reducing the amount of communications between tasks. When the number of processors is increased, the scheduling algorithms assign the tasks to more processors in order to better exploit the available resources. However, this increases interprocessor communications, and thus also communication overhead. Because GSM-LPT eliminates communication costs between tasks which are fused, it has a natural advantage over the straight LPT algorithm.

2. GSM-Bounded Optimal vs. Bounded Optimal

In addition to testing against the LPT algorithm, we chose to run one experiment using the Bounded Optimal algorithm, which returns the best schedule found after

enumerating a set number of schedules in the depth first search. We chose 10,000 trials as the bound to keep the run time reasonable. The next plot, Figure 22, shows the performance of both algorithms for the same set of 100 graphs:

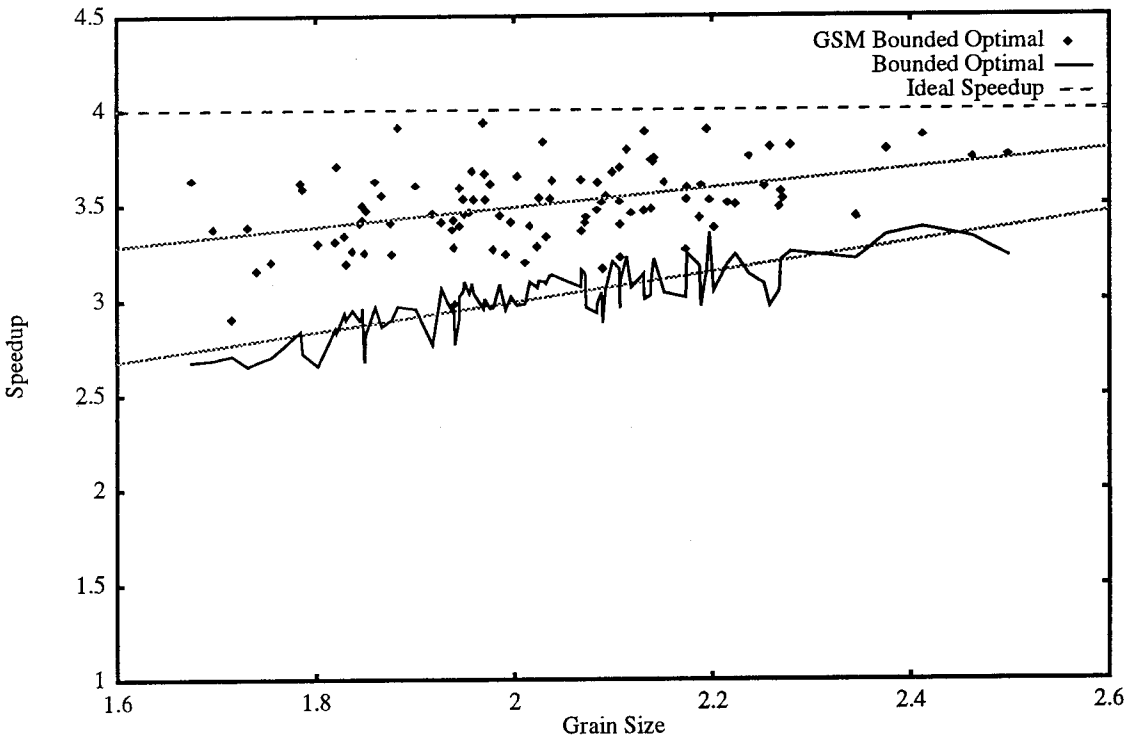

Figure 22: Effects of GSM on Bounded Optimal Algorithm, 4-Processor AN/UYS-2

The difference in performance is similar to that between LPT and GSM-LPT. This is probably because our implementation of bounded optimal starts from the LPT schedule, and 10,000 iterations does not comprise a large enough percent of the search space to find a significantly better schedule. Unfortunately, a more effective bound would take too long to run. This batch of 200 data points required five and a half hours of execution using seven Sun™ sparc 10 and three Solbourne™ S4000 computers in parallel.

3. Performance of GSM During Execution

We now look at run-time characteristics of the GSM algorithm. The first question is how quickly does the algorithm converge to a good solution? We explored this by recording the speedup and granularity of several task graphs during the execution of the GSM algorithm. Figure 23 shows the results for 14 task graphs taken from the original set of 100. Only the changes in granularity that were accepted - those that resulted in increased speedup - were recorded as a fission or fusion event. The results are shown below:

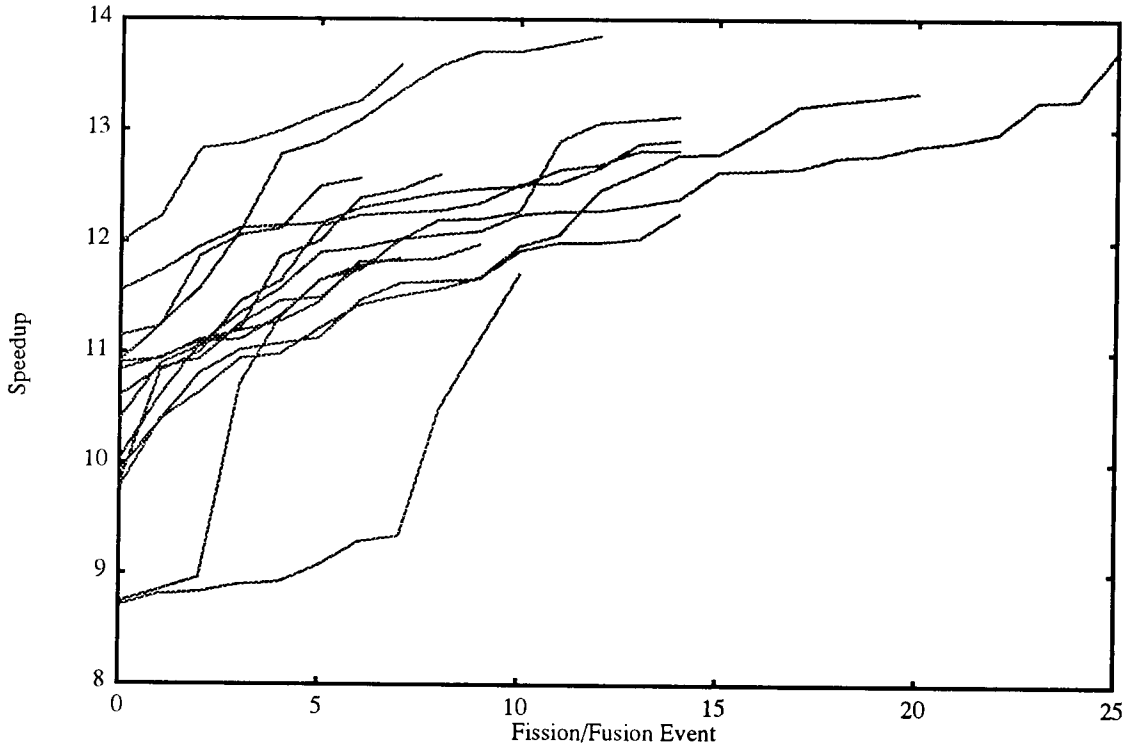

Figure 23: Speedup Change during GSM-LPT execution, 16-Processor AN/UYS-2

As can be seen, the increase in speedup does not decrease significantly during execution. Even more revealing is the low number of fission/fusion events. This indicates that the number of iterations should not be limited for the sake of efficiency.

The next plot shows changes in granularity for the same 14 graphs during execution.

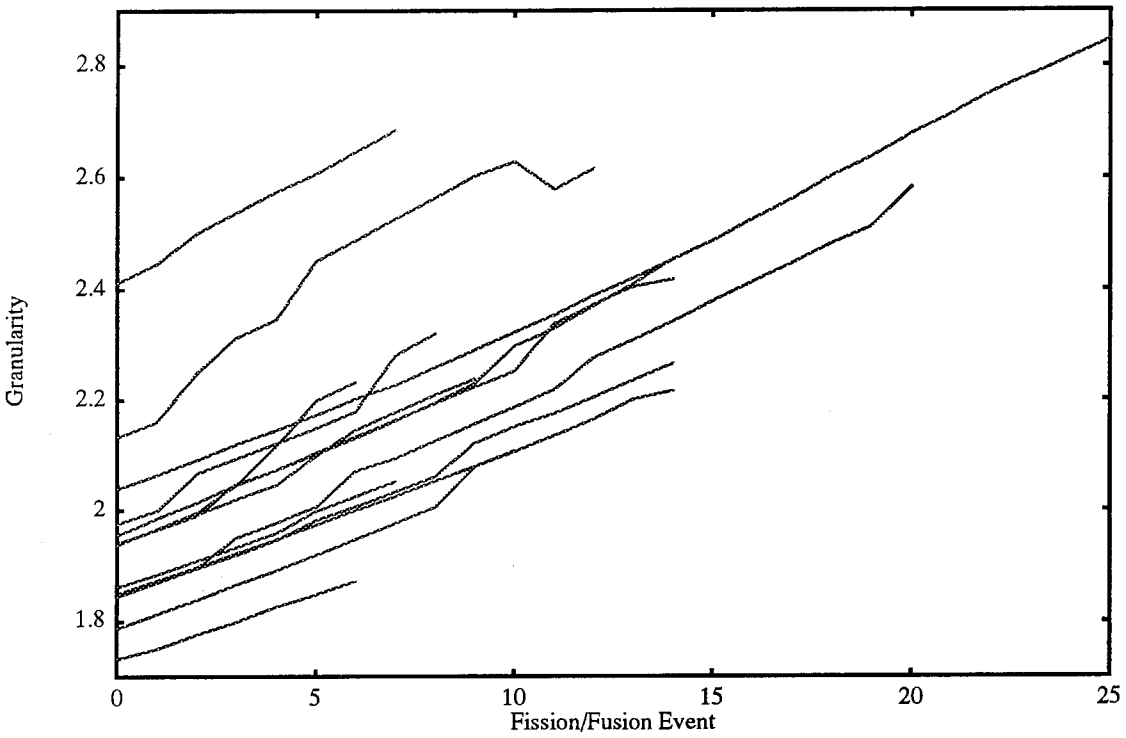

Figure 24: Granularity Change during GSM-LPT execution, 16-Processor AN/UYS-2

The almost constant increase in grain size can be attributed to the fact that fission is rarely needed when the number of tasks greatly exceeds the number of processors. The slow, linear nature of increase indicates that only a small number of nodes were clustered at any one time. This is important, since the data indicates a sudden increase in granularity can cause the algorithm to terminate. Since GSM is a simple greedy algorithm, sudden increases may indicate that a local minimum was found, thus halting the algorithm.

The last figure shows the average speedup at each grain modification event for each of the 14 speedup plots in Figure 23.

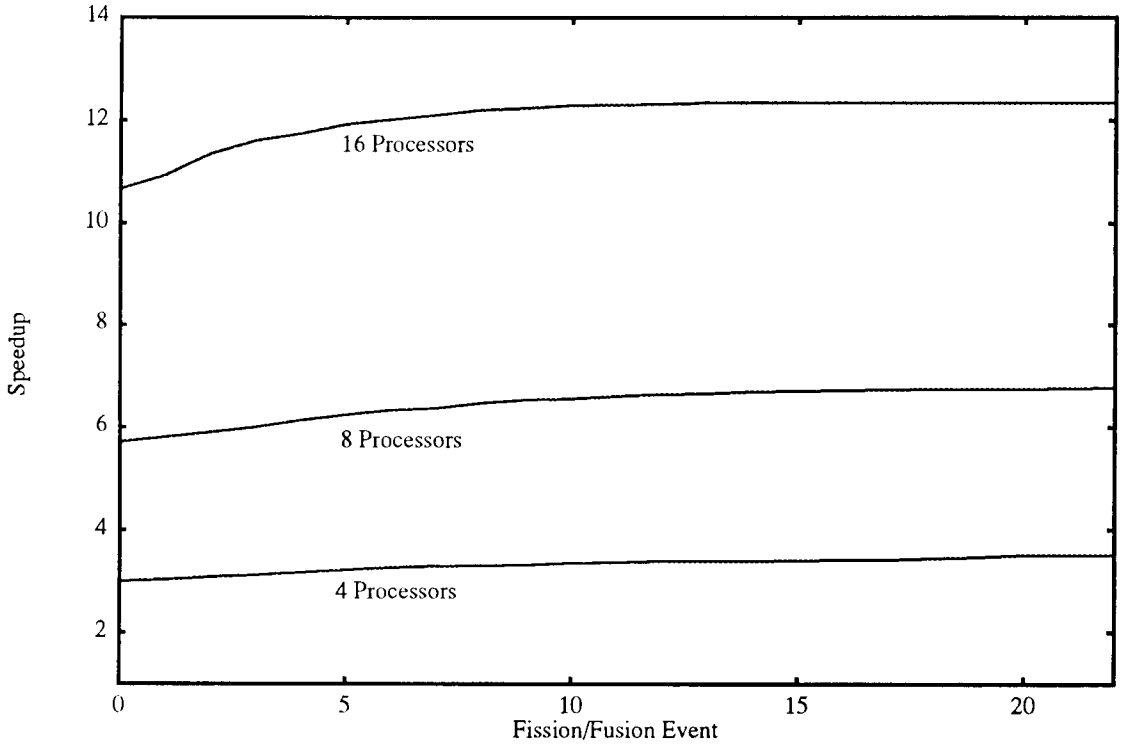

Figure 25: Average Speedup during GSM-LPT algorithm on AN/UYS-2

Although it is hard to see, the curve for the sixteen processor machine levels out sooner than for the eight processor curve, which in turn levels out before the four processor curve.

V. CONCLUSIONS

A. SUMMARY

We have shown that high throughput schedules can be obtained for multiprocessor computers by using simple grain size management techniques. The results clearly indicate that GSM provides a consistent improvement in throughput over unmodified algorithms for a variety of architecture configurations. Additionally, the GSM heuristic can be used with any scheduling algorithm that does not already employ grain size modification, and any scheduling problem where the computational requirements of the applications are known in advance, and can be expressed as acyclic task graphs.

Our results were based on the assumptions that tasks in a graph can be fused, and more importantly, fissioned, so that large nodes may be arbitrarily reduced. For a previously unfused node, we assumed fissioning resulted in two nodes that communicate with an amount of data equal to the sum of the original nodes' input and output amount. While this seems reasonable, it is an arbitrary assumption, and may be too conservative for some applications, and too liberal in others. Additionally, we conducted testing using a computer model that did not eliminate communication costs between tasks scheduled to execute on the same processor. Adding this capability would have reduced the effectiveness of our heuristic, which provided the sole means of reducing the amount of communications between tasks, although the overlapping of communication with computation in the model mitigated this somewhat.

B. RECOMMENDATIONS FOR FUTURE WORK

There are several aspects of this work that merit further research. The most fundamental is the issue of node fission. A more quantitative model needs to be developed that is more firmly grounded in the details of actual DSP tasks.

We also suggest that more testing of the GSM heuristic be conducted to determine its performance for a larger number of available processors. We did not use processor configurations which were large enough to demonstrate the merits of node fission, which

clearly needs to be done. Testing should also be expanded to cover the granularity spectrum from heavily communication bound to heavily computation bound graphs. There is also promise in trying different scheduling algorithms and computer architectures.

Further research could also be done to improve the search method, to enable searches past local minima. Obviously more sophisticated heuristics could be employed, but their advantages would have to be weighed against their higher time complexities, which must be combined with the time complexity of the scheduling algorithms as well.

Finally, research could be conducted to determine the cause and effect relationship between grain size and schedule performance. If a simple relationship does exist, then a new heuristic could be developed that does not require feedback from a separate scheduling algorithm to determine the suitability of the grain size. Such a heuristic would be more efficient than the GSM heuristic, and would give better insight to the scheduling problem.

LIST OF REFERENCES

[1] Karp, M. K., Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Math*, Vol. 14, No. 6, November, 1966.

[2] Garey, M. R., Graham, R. L., Johnson, D. S., "Performance Guarantees for Scheduling Algorithms," *Operations Research*, Vol. 26, No. 1, January-February, 1978.

[3] Hoang, P. D., Rabaey, J. M., "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput," *IEEE Transactions on Signal Processing*, Vol. 41, No. 6, June 1993.

[4] Norman, M. G., Thanish, P. *Models of Machines and Modules for Mapping to Minimise Makespan in Multicomputers.*

[5] Kernighan, B. W., Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell Systems Technical Journal,* February 1970.

[6] Thomae, D., *A Survey of Heuristic Partitioning Techniques,* 12 November 1993.

[7] Pothen, A., Simon, H. D., Liou K., "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM Journal of Applied Math*, Vol. 11, No. 3, July 1990.

[8] Chaudhary, V., Aggarwal, J. K., "A Generalized Scheme for Mapping Parallel Algorithms," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 4, No. 3, March 1993.

[9] Shen, C., Tsai, W., "A Graph Matching Approach to Optimal task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Transactions on Computers*, Vol. c-34, No. 3, March 1985.

[10] Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, 13 May 1983.

[11] Shukla, S. B., Little, B. S., and Zaky, A., "A Compile-time Technique for Controlling Real-Time Execution of Task-Level Data-flow Graphs," *Proceedings of the 1992 International Conference on Parallel Processing,* August 1992.

[12] Cross, D. M., *Usefulness of Compile-Time Restructuring of LGDF Programs in Throughput-Critical Applications,* Master's Thesis, Naval Postgraduate School, September 1994.

[13] Rice, M. L., "The Navy's New Standard Digital Signal Processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers 27th Annual Technical Symposium, 23 May 1990.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code CS 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

4. Prof. Amr Zaky, Code CS/Za 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118

5. Prof. Mantak Shing, Code CS/Sh 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118

6. Lt. Greg L. Negelspach 2
   5505 SE 22nd Court
   Gresham, OR 97080